Smarties: An Input System for Wall Display Development

Olivier Chapuis[1,2] **Anastasia Bezerianos**[1,2] **Stelios Frantzeskakis**[2,1,3]

[1]Univ Paris-Sud & CNRS (LRI)
F-91405 Orsay, France

[2]INRIA
F-91405 Orsay, France

[3]University of Crete
GR-70013 Heraklion, Greece

ABSTRACT

Wall-sized displays can support data visualization and collaboration, but making them interactive is challenging. Smarties allows wall application developers to easily add interactive support to their collaborative applications. It consists of an interface running on touch mobile devices for input, a communication protocol between devices and the wall, and a library that implements the protocol and handles synchronization, locking and input conflicts. The library presents the input as an event loop with callback functions. Each touch mobile has multiple cursor controllers, each associated with keyboards, widgets and clipboards. These controllers can be assigned to specific tasks, are persistent in nature, and can be shared by multiple collaborating users for sharing work. They can control simple cursors on the wall application, or specific content (objects or groups of them). The types of associated widgets are decided by the wall application, making the mobile interface customizable by the wall application it connects to.

Author Keywords

input toolkit; wall display; hand-held touch devices; cscw; multi-cursors.

ACM Classification Keywords

H.5.2 [Information Interfaces and Presentation]: User Interfaces - Graphical user interfaces

INTRODUCTION

High-resolution wall-sized displays allow multiple people to see and explore large amounts of data. They are well adapted to data analysis and collaboration, due to physical navigation that affords a natural pan-and-zoom in the information space, an enlarged physical space that enables collaborative work, and millions of pixels that allow viewing large amounts of data in one shared environment [1, 8]. They are well suited for application domains such as command and control, data visualization, astronomical imagery, collaborative design, etc.

Deciding on appropriate interaction techniques for wall displays is nevertheless not a simple matter. Mice, keyboards and direct touch are limiting in environments where more than one user can *move freely*, come close to the display to see details or move away to acquire an overview [1]. Research on mid-air interaction for remote displays (e.g. [22, 37, 25]), and recent work on mobile devices (mainly smartphones, e.g. [20]) focuses on specific interactions such as navigation, pointing and selection. Thus it cannot be applied as-is in real wall-display applications that need support for *multiple users* performing *complex interactions* that combine navigation, pointing, selection, dragging, text editing and content sharing. Finally, interaction techniques are often application or content specific (e.g. using a brain prop to rotate virtual brain scans [10]), requiring considerable design and implementation effort, thus making quick *prototype development and setup* challenging.

The few existing toolkits for programming collaborative interaction on walls require a significant effort to develop communication protocols between input devices and applications (e.g. [28]), prohibiting quick prototyping. Or assume users are static (e.g. [35]), forcing them to carry multiple devices (mice and keyboard) to perform complex tasks while moving.

The design goal of *Smarties* is to address all of the above: support complex interactions, using mobile devices to accommodate multiple mobile users, in a way that is easy to setup, develop, and use with different wall-display applications. Our motivation is the following: although specialized interaction techniques and devices can be very well adapted to specific applications, often wall application developers need input technology they can setup and use quickly to prototype and test their interactive applications.

Concept and Contributions

The components of the *Smarties* system, whose concept is described next, are: (i) an input interface on touch mobile devices (mobile input interface), (ii) a communication protocol between these devices and wall applications, and (iii) libraries implementing the protocol at the wall application side.

Mobile input interface

Classic desktops include a pointing device (mouse), a keyboard, and a clipboard to store data. If a large number of mice is available, together with associated keyboards and clipboards (we'll call them extended mice), we could use them for different *tasks* (e.g. one for pointing, one for selecting objects, or one for editing a shape or a text object in a drawing application). Or we could leave a mouse permanently attached to one or more specific objects (e.g. selected drawing shapes), making it synonymous to the objects it's attached to, i.e. a *shortcut* to them. In a simple desktop, if we copy a shape and then a text object in the same application, the second copy would overwrite the first. With the extended mouse idea both copies can still be available in their respective mouse's clipboard, ensuring *persistence* of interaction at the task level.

These extended mice, which we call *pucks* due to their round shape, form the central component of our mobile input interface. Each puck also has specific actions available to them, in the form of gestures or widgets on the mobile device: e.g. a

Olivier Chapuis, Anastasia Bezerianos & Stelios Frantzeskakis. Smarties: An Input System for Wall Display Development. In CHI '14: Proceedings of the 32nd international conference on Human factors in computing systems, 2763-2772, ACM, April 2014.

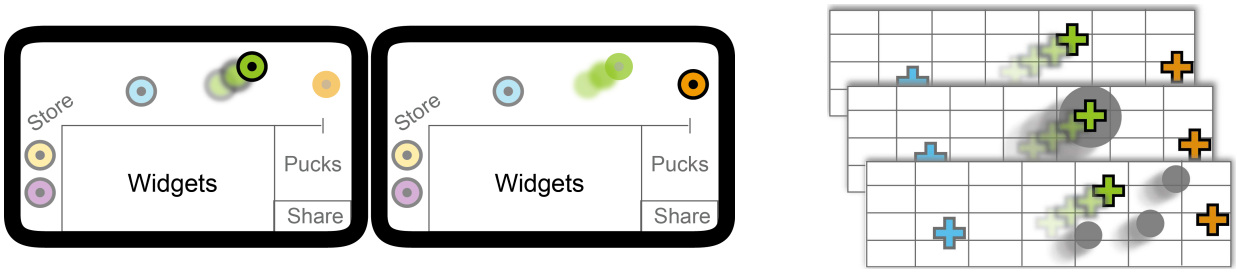

Figure 1. Left image: mobile device interface. The top area is taken up by multiple pucks. The bottom area has space reserved for (i) storing unused pucks; (ii) buttons for creating and sharing pucks; and finally (iii) an area for widgets *customized by the wall application* that can be associated to the active puck (green one here). The second image presents the view from another mobile device. Here the active puck is the orange one and the green one looks faded-out as it is unavailable. The last image shows possible presentation and behavior of the cursors on a wall display, controlled by these pucks. For example a moving puck can be associated with a simple moving cursor (top), moving an object (middle), or a group of objects (bottom).

puck attached to a text object can have shortcut action buttons for turning text bold, italic, etc. If we move this puck to another text editing object (even in another application) the same associated actions will still be available.

Multiple such pucks are available at any given time for performing different tasks, and their states (thus the users' work) can be *stored*. They can be also shared between multiple people to *share* tasks with colleagues: e.g. a user can hand over all her text editing work (including current mouse position, widget states, clipboard) just by handing over the puck.

This is the concept behind our interface: it is a collection of extended mice, referred to as pucks, together with their associated keyboards, widgets and clipboards. They reside in multiple touch mobile devices, ensuring that users can *move freely* in front of the wall, and control a wall application. They can be seen as simple mouse cursors, or shortcuts to specific tasks or content on the wall display. They are persistent in nature and can be shared among collaborating users. In our design, the associated widgets and keyboards are decided upon by the wall application, making the puck interface *customizable* by the wall application they connect to. See Fig. 1.

Protocol and Library

Smarties uses a client - server logic: the server is the wall application and the clients are the mobile devices. A protocol ensures the communication between mobiles and the wall application, with messages to: set up connections, maintain synchronization of pucks and their widgets/clipboards/keyboards across devices, and send high level input events (e.g. gestures or widget values) associated with the pucks.

The libraries implement the protocol in the server side and provides developers with the following functionality: a centralized way to synchronize pucks and their widgets across devices, ways of implementing ownership and locking of pucks, an event loop with callback functions to handle the events sent by pucks and their widgets, and methods for dealing with event conflicts from multiple devices.

The major advantage of the protocol and library is that the internal workings of pucks are hidden and developers can setup and use them as they would use regular mice and widgets. Thus, they provide a *quick* way of setting up and prototyping interaction support for wall display applications.

The main contributions of our work are:

- An open-source framework that combines (i) a mobile input interface, (ii) a communication protocol between multiple mobiles running the interface and the wall applications, and (iii) libraries encapsulating the protocol and mobile interface customization functionality, allowing for fast development of *input support* for collaborative interactive wall applications.
- The library hides completely the communication between wall application and mobile interface devices(s) from the developer. It provides collaborative interaction support in a few lines of code in the wall application side. And it allows the customization of the mobile interface with simple instructions in the wall application side, without modifying the code on the mobile devices.
- The mobile input interface components support complex interaction, from touchpad, keyboard, and clipboard, to combinations of specialized widgets such as menus, buttons or sliders with programmable interaction behavior (e.g. a button for "gathering" a set of selected objects). Multiple interaction elements called pucks act as shortcuts either to user tasks or wall display content, allowing for persistent work, that can can be stored and shared with other users.

Contrary to systems such as Pebbles [21], jBricks [28], ZOIL [17] and iRoom/iStuff [2], Smarties focuses on the input side only, offering an integrated system with a *high-level protocol* coupled with libraries that *hide the complexity of the protocol*, for quicker input prototyping. Contrary to these previous systems, it also comes with a ready to use (but customizable) *original input interface* running on mobile devices, handling advanced input (e.g. widgets and multi-touch gestures) and collaborative interaction (e.g. sharing policies).

SMARTIES MOBILE INPUT INTERFACE

The interface on the mobile clients is divided into two areas, the *touch* area where pucks exist and the *widget* area (Fig. 1).

Puck Visual Design and Basic Interaction

A proxy of the entire wall is represented visually on the top of the mobile device (touch area). A puck is represented as a small colored circle. We chose a round shape to both provide a large enough target and remind users of a touch footprint[1].

[1] Multiple pucks together look like Smarties candies, thus the name of the system.

Users can create several pucks on their device using the "Pucks" container on the widget area. Each device has at most one *active* puck at a time, rendered more opaque. Pucks can be deleted by moving them back to the "Pucks" area, or stored for later use by dragging them in a corridor on the left. Stored pucks can retain their interaction behavior and any properties the wall application associated with them. These designs were informed by user studies (see Applications).

A puck can simply control a cursor of the same color that appears on the wall. Moving the puck on the touch area moves the corresponding wall cursor in different ways. When users drag the puck itself, its wall cursor is moved with a direct mapping, traversing quickly large distances on the wall display. When users start the drag outside the active puck, we use an indirect mapping with appropriate CD gain transfer functions (see [24]) that slow down at low dragging speeds. This allows precise cursor movements even when the touch area is relatively small compared to the size of the wall. To allow switching between pucks, but limit accidental switching, users can long-press on another puck to activate it.

By default, a puck is visible in all mobile devices connected to the wall application to provide awareness during collaboration. An active puck on one device is seen as locked (faded out) on other devices and other users cannot use it. This puck becomes available to all users implicitly when it is no longer active, i.e. when the user selects another puck, or explicitly, through a "sharing" button. We have implemented alternative sharing policies described in the library section.

Widgets and Advanced Interaction

The widgets contained in the widget area are application dependent, and specified directly by the wall application without changing the mobile interface code (see library section).

A widget can control the active puck's behavior (e.g. a state button decides dragging vs hovering behavior for the corresponding wall cursor), execute actions (e.g. a button permanently attaches a set of objects to the puck), or control parameters (e.g. a slider changes the opacity of attached objects).

We currently support text view widgets, buttons, toggle buttons, check boxes, radio buttons, sliders, and different popup menu types. For example if users want to annotate objects attached to a puck, the wall application can specify a button "annotate" that pops up a keyboard and a dialogue window with a text field. When the user finishes typing and presses ok the text is sent from the mobile device to the wall application. We give more examples in the applications section.

By default a widget is puck dependent: its actions and values are associated to the active puck, and can thus change or even disappear when the user activates a new puck. However, a widget can be specified by the wall application as puck independent, for executing global actions, e.g. loading a new scene in our Lenses application example.

The system also supports several touch and tap gestures with single or multiple fingers. For example to allow wall applications to distinguish between cursor hover and drag, the touch area can distinguish a simple drag event (hovering) and a tap-and-drag gesture to emulate the usual press-drag-release interaction seen in touchpads. As we will see in the library and Lenses application example, detected multi-touch taps and gestures (e.g. multi-finger pinch or move) are not necessarily linked to puck movement and can be freely interpreted by the wall application for other purposes (e.g. zoom the wall view).

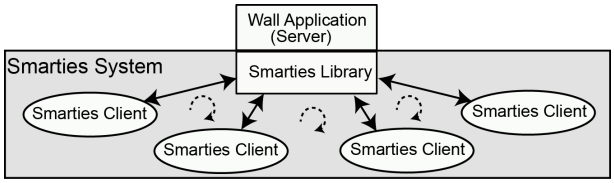

Figure 2. Basic Architecture. Rounded arrows indicate that the mobile clients communicate via the library, e.g. if a puck is moved in a client, this information is sent to the library that (i) transforms it into an event for the application developer; and (ii) sends it back to the other clients.

THE SMARTIES COMMUNICATION PROTOCOL

A common software architecture for tiled wall displays consists of a master application running on one machine (server), that may be connected to slave machines on a rendering cluster. User input is sent to the master application, that in turn instructs the slaves to modify their rendering state depending on the input events. A *Smarties* library sits on the master application side, managing input received from the mobile interface clients trough our communication protocol (see Fig. 2).

This abstract protocol is hidden by a library (described next), and ensures that the mobile interface client implementation is independent of the wall application. This section describes the internal communication process between the mobile clients, that are application agnostic, and the wall application.

Our high-level protocol: (i) is extensible, (ii) does not require programming or restarting the mobile clients, (iii) synchronizes states among multiple mobiles, (iv) supports complex multi-finger input and (v) widgets and keyboard mapping.

Extensibility

For the mobile clients a server is an IP address and a communication port that sends or receives messages. All messages from a mobile client to the wall server start with the IP address of the client, considered as their unique identifier. A *message* consists of a name followed by a sequence of typed values (boolean, integer, float, double or string), whose length depends on the name of the message (e.g. `<IP, menu, [list of item names]>` for a popup menu).

Our protocol builds upon OSC[2], a low level communication protocol that is flexible in message naming and length. It can thus be extended either by adding new messages or appending new values at the end of existing messages, ensuring that new types of widgets and behaviors can be added.

Mobile Client Customization

To connect to a wall server, a client sends a `NewConnection` message (msg) at startup or when the user changes the IP or port (i.e. the wall server ID). Clients send continuously interaction messages. Whenever the wall receives a msg from an unknown client it sends a `Hello` msg, and *whenever* a client

[2] **http://opensoundcontrol.org/introduction-osc**

receives such a msg it resets itself to receive customization information. Thus, it is never necessary to restart a mobile client (even if a different wall application is started) and a wall server can ask a mobile client to reset itself at any time (e.g. to install a different interface on the mobile).

After communication is established, the server initializes and customizes the mobile client. This consists of: (i) a msg defining default behaviors, e.g. what touch events the client should send; (ii) a description of the widgets that will appear in the widget area, their types, relative positions, values, labels, etc.; and (iii) the description of any existing pucks, through a series of `NewPuck` msg, consisting of a unique puck id, a position, a color, an icon name and a status (free/locked/active).

Puck Synchronization

Mobile clients ask the server to create a puck with a `AskNewPuck` msg. The server responds with a `NewPuck` msg with a unique puck id to all the clients (with active status for the requesting client). After that, to ensure interactive response times, the mobile client can update its pucks' state, and simultaneously send messages to reflect user interaction that modify the status of a puck (e.g. store, activate, move, etc.). In turn, the server forwards this information to the other clients, or can chose to ignore them and force a change of state on the requesting client. Thus while puck creation and management is centralized on the server side to synchronize different mobile clients, requests from mobile clients are also treated locally to ensure quick responses to users' actions.

Single- and Multi-touch events

Our protocol distinguishes one finger drag on the touch area used to manipulate the pucks (move, activation, etc.) from multi-fingers gestures and multi-taps that a wall application can use for specific purposes. We provide two alternatives (chosen by the wall server at connection time): a raw protocol that simply forwards the touch events (with time stamps), and a *Smarties* protocol consisting of higher level events.

The *raw* protocol sends the usual three events: `Down` or `Up` with a unique "finger" identifier and position, or `Motion` as an array of positions with a unique identifier for each down "finger".

The *Smarties* protocol sends msgs consisting of single and multi-tap events[3] that report: the number of taps and number of fingers for each tap, followed by single- or multi-finger move or multi-finger pinch gesture events. So a simple single-finger drag can be interpreted as cursor hover, while a tap and then drag as a press-drag-release interaction. Or a two finger pinch can be interpreted as global zoom, while a three finger pinch can scale a particular object. Thus due to the nature of the protocol, either the number of taps or the number of fingers can act as modifiers for the semantic of a gesture.

Widgets & Keyboard

When users interact with a widget on a mobile client, a msg is sent describing the id of the active puck and the new state of the widget (e.g. button click, state of a toggle button, value for a slider, etc.). The server propagates the msg to the other clients, synchronizing the widgets' state. For example, if a client changes the value of a slider, the server communicates this value to all other clients that in turn update the value of the corresponding slider immediately, if the slider is global, or when the associated puck becomes active on them.

Finally there are messages to ask a client to map or unmap a keyboard. Regarding key events (up and down) sent by the mobile clients, we have fixed a keyboard mapping so that the protocol does not depend on a specific client toolkit or OS.

[3] Sequence of finger taps separated by less than 200 ms.

SMARTIES LIBRARIES FOR WALL APPLICATIONS

We wanted *Smarties* mobile clients (under Android), to be setup and used as input by wall application developers, almost as easily as desktop developers can use a mouse. To simplify the protocol, a library implementation takes care of issues not directly related to the behavior of a wall application, such as connections, maintaining states, etc. We developed a multi-platform C++ library (libSmarties) and a Java library (javaSmarties) for *Smarties*.

The libraries hide the protocol and the communication needed to keep puck properties synchronized across mobile clients. It also simplifies the initialization, widget creation and handling through callbacks. The heart of the libraries is an event queue that provides *Smarties* events of various types: puck create/delete/store/activate, touch events and widget use. These come with data structures and classes for the pucks, *Smarties* events and widgets. The class for pucks also includes an object (the "clipboard") used solely for storing application specific data. Functions are also provided to facilitate the synchronization of widget states, and to access a large part of the protocol allowing customization, extensions and advanced use.

Wall application developers can create new sharing policies or use one of the three already implemented: *strict*, where pucks are unlocked and available to others only when an explicit share action is taken; *medium*, where a puck is immediately unlocked when another is selected; or *permissive*, where a puck is unlocked if it is not actively used.

Example walkthrough

Let us sketch the needed code for a *wall application* to support multi-cursors with pick-and-drop of graphical objects using *Smarties*. We use the C++ version of the library here, but both are (intentionally) very similar.

We first create a Smarties object with the wall geometry:

```
Smarties *smarties = new Smarties(wallWidth, ...);
```

We can then override some defaults, e.g. the sharing policy and the type of multitouch events, using simple class methods. The final step in the initialization is to create some widgets in the widget area. Here we create a slider in the center of the widget area to change the size of the cursor associated to the active puck, set the default value of the slider, specify that it is puck dependent (default) and attach it to a callback function.

```
SmartiesWidget *slider; int wid;
slider = smarties->addWidget(
      &wid, SMARTIES_WIDGET_TYPE_SLIDER, "Cursor Size",
      0.3f,0.3f,0.3f,0.6f);
slider->slider_value = 50; // default range from 0 to 100
slider->dependence(SMARTIES_WIDGET_DEP_PUCK); // default
SET_CALLBACK(slider, &sliderHandler);
```

After the initialization, the `smarties` instance is ready to run on a thread, `smarties->run()`. The library provides access to the events that can be handled in a classic "event loop":

```
SmartiesEvent *evt;
while((evt = smarties->getNextEvent()) != NULL) {
  puck *p = evt->puck;   // the puck of this event
  float x =  (p->getX()*wallWidth);  // x pos in the wall
  float y =  (p->getY()*wallHeight); // y pos in the wall
  // switch on event type ...
  switch(evt->type) {
    case SMARTIE_EVENTS_TYPE_CREATE:
      // a new puck; create an associated WallCursor
      p->app_data = new WallCursor(x, y);
      break;
    case SMARTIE_EVENTS_TYPE_DELETE:
       delete (WallCursor)p->app_data; // remove wall cursor
       // allows the library to delete the puck
       smarties->delete(p);
       break;
    // ... handle the other event types
```

In the code above we assume that we have a `WallCursor` class that draws a cursor at a given position, and the code just (i) creates an instance of this class for each new puck and stores it in the field of the puck object reserved for the application; and (ii) removes the wall cursor if the puck is deleted. Store and restore puck events can also be handled by using methods defined in the `WallCursor` class.

We assume that the application has a picker to select graphical objects rendered in the wall, and that such objects can be attached to a cursor. Here is an example of coding pick-and-drop interaction (tap to pick) using *Smarties* touch events.

```
    case SMARTIE_EVENTS_TYPE_TAP:
        WallCursor *wc = (WallCursor)p->app_data;
        if (wc->attached_object != NULL) {
            wc->attached_object = NULL; // drop
        else { // pick eventually
            wc->attached_object = pickObject(x,y);
        }
        break;
    case SMARTIE_EVENTS_TYPE_MOVE:
        // move wall cursor and attached_object if not NULL
       ((WallCursor)p->app_data)->move(x, y);
       break;
```

Widget callback functions are called in the same manner from the event loop, for synchronization purposes and for allowing to pass on data depending on the interaction context:

```
    case SMARTIE_EVENTS_TYPE_WIDGET:
       evt->widget->handler(evt->widget, evt, some_data);
       break;
```

In our example, the slider callback just calls the `setSize` method of the `WallCursor` class that changes the cursor size:

```
void sliderHandler(
  SmartiesWidget *w, SmartiesEvent *evt, void *user_data) {
   WallCursor *wc = (WallCursor)w->puck->app_data;
   wc->setSize(w->slider_value);
}
```

The example illustrates how implementing mobile multi-user input for a wall application with libSmarties resembles closely the usual development of interactive applications using an event loop. Here multi-user pick-and-drop is supported with code very similar to the one a developer would use to code pick-and-drop for a single mouse. Thus, *Smarties* allows to quickly prototype input for mobile multi-user interaction, so as to move fast into more interesting aspects, for instance collaborative pick-and-drop: observe how one user can pick an object with a puck on one side of the wall, share it with another user, that can then drop it on the other side.

Although the library treats commands executed simultaneously as FIFO (e.g. when two users want to activate the same free puck), it does not deal with complex operations that may cause conflicts in the state of the application, for instance if a user tries to pick a graphical object that is already picked by someone else in our example. These situations are highly application dependent and as such need to be handled by the wall application itself, e.g. by adding a picked state to graphical objects that is checked in `pickObject` for our example.

APPLICATION EXAMPLES

Besides toy examples for testing, we developed three wall display applications to demonstrate our framework. These server applications are developed in different rendering engines, a Java one (zvtm-cluster [28]) and two C++ ones (Equalizer [7], and Qt[4] with OpenMPI[5]), showing how *Smarties* is independent of the rendering engine. The first application was used to design the *Smarties* concepts and client interface, informed by user studies. The other two use libSmarties and are drastically different, demonstrating the generality of the *Smarties* system.

a. Object Grouping (server in ZVTM cluster, Java)

In a workshop we conducted on potential wall display uses, a group of biologists felt wall displays could be an appropriate environment to collaborate for their task of cataloging photos of plants sent by volunteers in the field. Depending on their expertise, they sort and tag the images based on specific characteristics (origin, leaf or stem shape and color, flower family, etc.), compare them with existing images for similarities, and group them into entries of existing plants. Similar needs for wall display use have been identified in [10] where a team of neuroscientists needed to compare and classify brain images to study variations in the brain.

Motivated by such scenarios we developed a prototype application (*Smarties* client and wall application), that allows users to access one or more objects on the wall display, apply properties (tagging), and perform actions on them (grouping and moving). The prototype can be seen in Fig. 3 and was preceded by two iterations used to run laboratory experiments.

Interface: A puck's behavior is set through toggle buttons on the widget area: (i) a select mode adds or removes objects in a group associated with the puck by a simple tap when the corresponding cursor is on the object; (ii) a move mode where a simple gesture on the touch area moves together all the objects selected by the puck; and in our final prototype (iii) a cursor-inside mode that allows interaction inside an object as if it is a classical desktop application window. In this last case the cursor associated to the puck is confined inside the object and the touch area of the mobile devices acts as a touchpad (in our prototype we use it to treat some objects as post-it notes where free hand drawing and annotation is possible).

[4] **http://qt.digia.com/** [5] **http://www.open-mpi.org/**

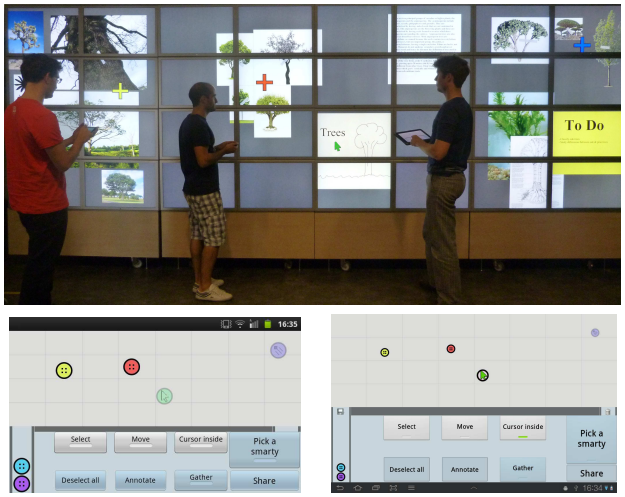

Figure 3. Object selection and grouping with three users. The interface running on a *phone* (left, middle user) and on a *tablet* (right, right user). Six state button widgets have been added by the application. The "Cursor Inside" action (device on the right) is attached to the active green puck, and acts as a mouse cursor confined inside a window.

The *Smarties* widget area also has buttons to perform actions: "Gather" groups spatially all the selected objects of a puck; "Deselect All" deselects all objects selected by a puck; and "Annotate" pops up a keyboard and adds a text tag to the objects selected by a puck. These behaviors and actions indicate how a puck goes beyond cursor control and can be associated with multiple wall objects and properties.

User Studies: With an initial prototype, we run a first user study comparing *Smarties* to an interface where objects on the wall were represented on the client device, a simplification of a world in miniature (WiM) interface [34]. Participants in pairs, had to group rectangles on two different locations on the wall either based on their color or on a small text label that forced participants to see rectangles up-close (see Fig. 4). We varied the difficulty of the task by controlling the number of rectangles to be classified (10 or 30) and by optionally adding distractor rectangles (0, 10 or 30) using a third color or label.

We found that *Smarties* (i) leads to fewer input conflicts, i.e. manipulation of the same object by two user (a conflict happened in 1.9% of the trials for *Smarties* and in 8.6% of the trial for *WiM*); (ii) leads to fewer errors, i.e. moving an object in the wrong place (error rate of 5.3% for *Smarties* and of 14.3% for *WiM*); (iii) showed better performance when tasks become more difficult (presence of distractors and number of objects). Moreover, participants gave it significantly higher subjective scores on speed, accuracy, comfort and cognitive demand. We also noticed that participants could use the touch area of the *Smarties* client while keeping their attention on the wall.

In a second experiment we explored how users shared and reused pucks, refined their selections, and tagged sets of objects. The task was motivated by our biologists scenario: participants had to select groups of objects and then progressively refined these selections depending on different roles assigned to them. Optimal strategies led to exchanging selections by sharing pucks and keeping some selections alive

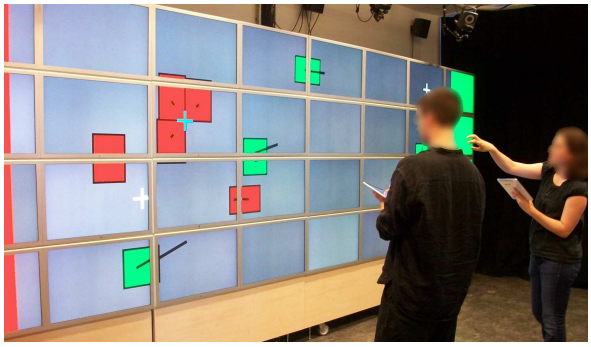

Figure 4. Participants collaborating and performing grouping by color.

for reuse. Overall, pucks helped users share and exchange work. Nevertheless, their reuse comes at the cost of display clutter: users either kept potentially useful pucks by placing them out of the way; or decided to continue using them for other tasks to avoid having many pucks on the screen, risking duplicating work later. This led to the design of the "storage" pucks area on the left of the widget area, that allows short term storage (persistence during a working session). Application programmers can turn this into a long term storage by saving the state of the stored pucks and their widgets, and resending them to mobile clients at connection time in the next session.

b. Multiple Lenses and DragMags (server Equalizer, C++)
Despite their size and resolution, wall displays are relatively small compared to existing data sets and big high resolution images (e.g. galaxy surveys). This led to the study of pan-and-zoom navigation alternatives [25]. However, these techniques are not well adapted to multi-user contexts, as they affect the entire screen and prevent concurrent navigation. We developed a prototype (Fig. 5) where users can create and use several fisheyes, magnification lenses and DragMags to navigate scenes, allowing local multi-user multi-scale navigation. We used Equalizer, a powerful platform for developing cluster applications for intensive, but fast, graphics rendering.

Interface: When created, a puck is a simple touchpad cursor on the wall. Actions are performed on the object (lens or anchor) that is each time under the puck cursor. Users can use a button on the widget area to create a magnification lens at the cursor's position. They can then change the type of lens with a popup menu (magnification, magnification with transparency, or fisheye), transform a magnification lens into a DragMag (a lens whose focus or "anchor" is at a remote location), or move it to a new location using a tap-and-drag gesture.

A puck can also be attached to a specific object with a toggle button, and any subsequent puck movements move the attached object. This is interpreted by the wall application as locking the object to that puck, making it unaccessible to other pucks. Thus lenses can act as territories that mobile users can lock and move with them.

A DragMag can be manipulated by two pucks, one attached to the lens itself and one to its anchor. Users can move the anchor around to see content from different areas of the wall close to their position (as in [5, 19]). This can be done collaboratively by two users, each manipulating one puck.

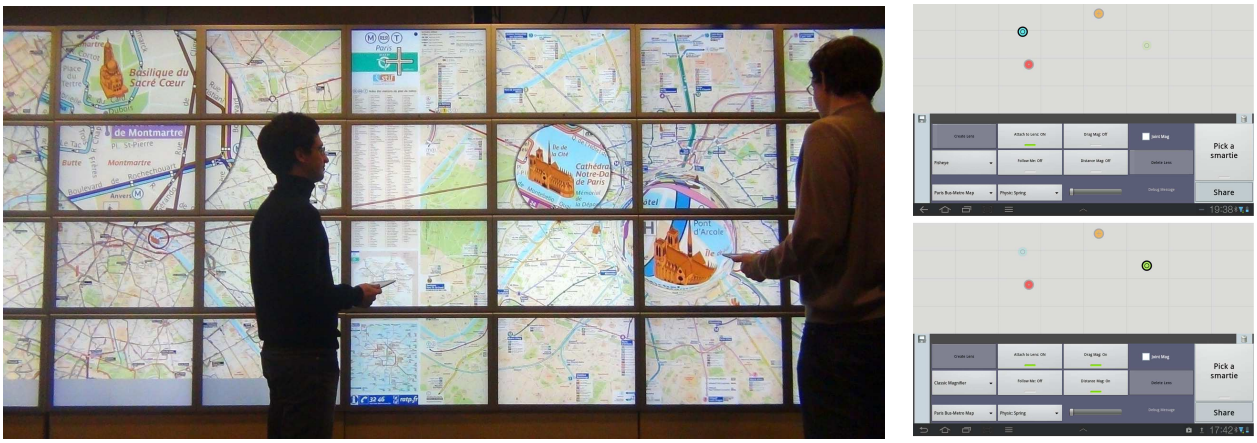

Figure 5. Left: Multiple Lenses, starting from the left a magnification lens, a DragMag and a fisheye. Right: the two mobile client interfaces running on tablets. The four pucks are attached respectively to a magnification lens (left of wall), the anchor and lens of a DragMag (middle) and a fisheye (right). The active puck is the blue for the device on top, and green for the bottom. The described widgets added by the application are seen on the widget area.

We grouped global widgets (i.e., independent of the active puck) at the bottom of the widget area. A drop down menu loads a new scene, and another sets the behavior of two lenses bumping each other. We use a 2D physical model where the lenses are considered as disks with their center attached to the ground with a spring. A global slider is used to change the strength of the spring from rigid, where bumping lenses don't move at all, to flexible, where they are pushed out of the way by other lenses and spring back when possible.

Advanced Functionality: Thanks to libSmarties it was easy to add multi-touch features: A two finger pinch changes the magnification factor, or resizes the lens if it is preceded by a tap. A three finger pinch changes both the size and the magnification factor so that the content rendered in the lens does not change. We also use a five finger pinch for lens creation (finger expansion) and deletion (finger contraction).

We used the store area to "bookmark" positions on the scene. Dragging a puck attached to a lens into the store area will hide the lens, but the wall application remembers both the position and the properties of the lens (type, magnification factor, etc.). If a user restores the puck, the wall application restores the lens and its properties to its original position.

Input Extensions: Although we developed this application using *Smarties*, we easily extended it to other input techniques, such as implicit input by tracking user movement. We used the distance of the user to the wall to change a lens' magnification factor (keeping the viewing area constant), and the position of the user to have the lens follow her. This required some setup work (linking the prototype to a motion tracking system and code appropriate computations). When it come to the user interaction, i.e. to allow a user to enable and disable these features, we added 10 lines of code to the wall application: two toggle buttons to the clients and corresponding widget handlers that enable/disable the features related to the different input techniques. By following a MVC development architecture we were able to easily share the event handling code from *Smarties* with other inputs.

c. Wall Native Cursors (server in Qt with OpenMPI, C++) Several pieces of software[6] allow sharing a mouse, keyboard and clipboard between computers. In the presence of a rendering wall cluster, one can use the mouse and keyboard of a computer outside the cluster to control different machines of the cluster by moving the cursor from the "edge" of one screen to another. This allows interacting with the native window system of each machine for testing or admin purposes.

Interface: We implemented such a software on top of *Smarties* and extended it further. An active puck moves the native cursor of the screen it is on, and the client's touch area is used as a touch pad: one finger tap is a left click; two and three finger taps are middle and right clicks; tap and drag are a press and drag; and two moving fingers emulate a mouse wheel (with four directions). For text input the clients contain a popup keyboard, and buttons that emulate complete keyboard shortcuts that appear often (e.g. CTRL+Z). In the wall application, we forward the pointer/key events to the slave machines, and we added a transparent overlay covering each screen to render large cursors at the position of the pucks (larger than the native cursors), that are visible at a distance.

There are obvious advantages of using the *Smarties* over a traditional sharing application on a laptop or desktop computer: users can move freely and at any distance in front of the wall with their mobile, while "moving" interactively the cursor from screen to screen. They can also create several pucks, reserving some for certain areas of the wall, or associating them to some selection (as each puck has its own clipboard). Moreover, several users with their own pucks can interact in front of the wall, working with the native windowing system of the screens closest to them, thus transforming the wall into a computer lab. Users can also share their work via the clipboard by exchanging pucks. Note, however, that as our native window manager does not support multi-pointers, if two (or more) active pucks are on the same screen, then they will all send pointer events leading to cursor jumping. This can be solved by introducing a priority rule in the server side such as "the older puck in a screen controls the cursor".

[6] E.g. PointRight [18] or synergy `http://synergy-foss.org/`.

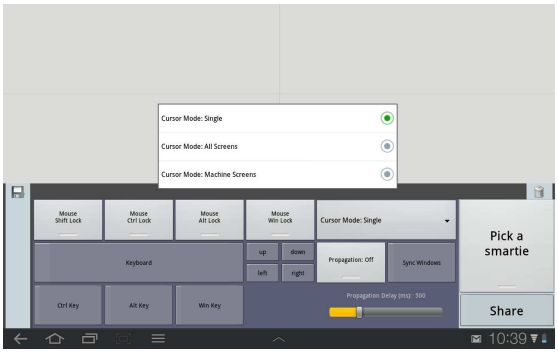

Figure 6. The mobile client running with the *Smarties* wall cursors application. A drop-down menu has been triggered by the top-left button of the widget area, to choose one of three replication modes.

Replication Extension: What came naturally to mind when testing this prototype was to replicate the interaction done in one screen to others. Our wall consists of 16 identical machines, each with two graphic cards driving two screens (30" and 2560×1600 each). We added a drop-down menu to the client (Fig. 6) for choosing ways to replicate interaction done in one screen with a puck: (i) no-replication (as the examples so far); (ii) all-machine replication (i.e., on half of the screens); and (iii) all-screen replication.

Replication can be used for graphical administration of a cluster. We configured the window manager (FVWM) so that a right click (3 fingers tap) pops up a menu of common applications (a terminal, packages manager, a browser, etc.). A user can then open 16 terminals simultaneously (one on each machine) and start typing commands. She can start the package manager and install an application on all the machines as she would on her desktop computer (we installed Gimp and VLC this way), or start simultaneously a web browser to search and download a video on all machines (as we did with the basketball clip shown in our video figure).

We used this replication mechanism to experiment with a collaborative artistic performance scenario using the image editor Gimp[7]. In all-screen mode an artist starts Gimp and draws simultaneously on all 32 canvases. The artist can switch to no-replication mode to draw on a specific canvas, or collaborate with other artists, each using their own puck in no-replication mode. At the end the artist saves the 32 created images in machine-replication mode and uses a script to upload and combine all the images into a single art piece.

We also experimented with a propagation delay in the interaction replication. We added to the *Smarties* clients a toggle button to switch on/off this propagation delay, and a slider to set the delay d. When this mode is on, the interaction performed by the puck on one screen is replicated to its adjacent screens after d ms, to their adjacent screens after $2 \times d$ ms, and so on. This leads to interesting effects, such as being able to observe one's interaction history.

Although this example started as a basic cursor support for the wall, the ease of programming and flexibility of *Smarties* allowed us to think of creative ways to interact beyond what we originally envisioned.

[7] `http://www.gimp.org/`

DISCUSSION AND FUTURE WORK

As Greenberg [11] explains, building toolkits is an iterative process. *Smarties* went through at least three major development iterations (see Applications), that were informed by usability studies of the interface and observations of its use in complex tasks, while progressively hiding nonessential housekeeping and input managing tasks. It has several desired characteristics of groupware toolkits [11], such as the ability to work in common languages (C++, Java) and use a known event programming paradigm, hide low level implementation details related to communication and sharing, and can be used in a few lines of code. More importantly, the flexibility of the library gave the means to think creatively and come up with diverse and often playful application examples.

As researchers in wall display interaction and visualization, it gave us the freedom to:

- quickly setup and run pilot studies to determine detailed interaction needs and explore possibilities for applications and systems we develop;
- easily prototype and run studies where the main focus is not the interface design (e.g. studies on visual perception), without worrying about interaction choices and mobility constraints (e.g. how can we have moving participants that need to provide answers both by typing and pointing);
- conduct iterative interface design, by progressively discovering interaction needs, and slowly replacing functionalities developed for *Smarties* by other interaction means (e.g. the addition of motion tracking in the Lens example).

As Olsen mentions [26] there are several ways to evaluate a toolkit. We hope that our application demonstrations showed the generality of *Smarties* for wall display input support. Our system is designed to reduce the effort of adding input control and goes beyond previous work on wall display input support, particularly during the early stages of wall application development. Future work includes a more thorough evaluation of our system by observing its use by other developers. We will also extend the *Smarties* (Protocol & Libraries) to support complex Multi Display Environments (e.g., several walls and tabletops) and geographically distributed settings. Currently developers can customize the mobile *Smarties* interface using predefined widgets and events. We plan to extend our system to allow developers to create their own widgets and gestures.

Beyond the toolkit extensions and evaluation, we will study the potential of the *Smarties* interface as more than a prototyping input mechanism. This requires studies comparing this interface to other interaction techniques adapted to mobile settings, such as laser pointers mounted on mobile devices, mid-air gesture input, and direct touch input on the wall. In this context it is important to understand the cost of attention switching between the wall and the mobile when users manipulate widgets on the *Smarties* interface. Finally, we will investigate how the *Smarties* interface, that has a moveable ownership context (e.g. groups of objects attached to a moving puck), affects the perception of working territories [32], spatial separation [36] and coordination [23], and coworker and workspace awareness during collaboration.

RELATED WORK

There is a large body of work on wall display interaction techniques using touch or a pen to reach and manipulate remote content (e.g. [9, 4]), accessing and manipulating content using pointing (e.g. [22]), freehand gestures (e.g. [25, 37]), or combinations of pointing and mobile devices [25]. There is also work on using custom interaction props (e.g. a motion tracked brain prop to rotate virtual brains [10]), simple physical devices [2] or tangible widgets attached to a tablet [16]. This work often requires specialized hardware (e.g. markers, devices, or touch enabled walls), or at the very least a setup, training and calibration phase.

Mobile devices such as smartphones and tablets, are widely available and familiar to users, and have been used as input for remote displays. Although there are several techniques that use the mobile's camera to interact with large displays, they often require visual markers to identify the remote display (e.g. [30]). Recently, Touch Projector [6] allowed interaction via a live video captured with the mobile's camera, without the need for markers on the remote screens. This work generally requires holding the mobiles at eye level to look at the remote display through their camera, and is better suited for brief interactions and not long term use.

Touch screens of new generation mobiles can be used to interact with wall displays without the need of additional tracking technology. They allow user mobility, while having a large enough interaction surface to accommodate more complex input. In Hachet *et al.* [13] multiple users can see in their devices a view of a 3D object displayed on the wall. Olwal *et al.* [27], use mobile devices to display and interact with parts of radiology material projected on a larger display. They support multi-touch navigation gestures for manipulating content. These approaches, following the idea of the peephole displays [39], assume that the mobile device is aware of the content displayed on the wall, or can at the very least render part of it. On the other hand, the Overlay interface [31] uses the touch display only as input by defining interaction areas for each user on a wall display. Similarly, ARC-Pad [20] uses only the mobile's touchpad to combine absolute (tap) and relative (drag) cursor positioning on a wall display. *Smarties* falls in the middle: our touch area has pucks representing links to display content (but not the content itself), and it is configurable while being application agnostic. It is closer to older approaches using PDA's, that treat the mobile device as a personal tool pallet (e.g. [29]), or as cursor and keyboard controllers (e.g in Pebbles [21] discussed later).

The majority of this work on interaction, supports only very specific tasks (e.g. point and select, pan and zoom), and needs to be re-thought in a fully operational environment where long term use may be tiring, text needs to be entered, and the wall includes interactive application windows [10, 38] and not simple targets. The rest tend to be complex to implement and are highly application dependent. Our goal is to provide a means to easily add complex interactive support to wall applications that is easy to setup and use for prototyping.

Existing toolkits for developing interaction on walls focus on other aspects. The SDG Toolkit [35] and nowadays native operating system support for multi-user interaction (e.g. [14]), focus on managing classic input devices attached to a screen (mice, keyboards, touch input). Supporting user locomotion is challenging, even when we consider carrying mobile versions of these devices, as in real life users switch frequently between tasks that require different devices. JBricks [28], ZOIL [17] and iStuff [2] provide customizable bridges between remote displays and different possible input devices, but require programmers to define or use low level communication protocols to treat the input events.

Pebbles [21], although not a toolkit, is a concept close to our work: it includes two different mobile applications, one that sends cursor and keyboard events from a PDA to a remote machine and one that provides widget controllers. *Smarties* goes further: it is a development toolkit that requires programming only on the server side (not the mobile), with few and simple lines of code for communication, and thus allows quick input prototyping of collaborative apps. It has a larger input vocabulary (gestures and widgets), creates shortcuts to content on the wall beyond simple cursors, and allows storing and sharing of interactive work between users.

Work on Single Display Groupware (SDG) [33] has investigated the effects of such environments on user behavior (e.g. [36]), problems both in following fast moving cursors [3], and multiple cursor awareness and identification (e.g. [15]). As pucks are often represented as cursors on the wall, this work has influenced some of our designs (for example different colored cursors [12]). These important research directions are orthogonal to our work, as we focus more on the control side (puck UI), but they need to be considered on the server side in real world applications that go beyond prototyping.

CONCLUSION

Smarties is an input system for wall sized display application prototyping. It consists of an application agnostic client that acts as the input interface and runs on multiple mobile devices, a communication protocol between the clients and the wall application, and a library that implements the protocol and handles input management.

The mobile application is made up of multiple interactive pucks and associated widgets (e.g. buttons, sliders, menus, text fields) that allow for command activation, and for changing properties of the wall application or of the puck behavior. A puck can be associated with content on the wall display (cursors, objects, groups of objects), and users can store and share pucks and thus their interaction work. Each wall application can customize a puck's widgets to fit its particular needs. A few lines of code initialize and setup the *Smarties* and widget management using an event loop and callback functions.

We demonstrated through 3 application examples using *Smarties* how the system supports very different wall applications, with different interaction needs, developed using different wall display software technology. We hope the ease and flexibility of *Smarties* will help wall application designers to quickly add mobile multi-user interaction support to their systems.

The *Smarties* software is available at **`http://smarties.lri.fr/`** under free software licenses.

REFERENCES

1. Ball R., North C. & Bowman D. A. Move to improve: promoting physical navigation to increase user performance with large displays. *CHI '07*, ACM (2007).
2. Ballagas R., Ringel M., Stone M. & Borchers J. iStuff: a physical user interface toolkit for ubiquitous computing environments. *CHI '03*, ACM (2003).
3. Baudisch P., Cutrell E. & Robertson G. High-density cursor: a visualization technique that helps users keep track of fast-moving mouse cursors. *INTERACT '03*, IOS (2003).
4. Bezerianos A. & Balakrishnan R. The Vacuum: facilitating the manipulation of distant objects. *CHI '05*, ACM (2005).
5. Bezerianos A. & Balakrishnan R. View and space management on large displays. *IEEE CGA 25*, 4 (2005).
6. Boring S., Baur D., Butz A., Gustafson S. & Baudisch P. Touch Projector: mobile interaction through video. *CHI '10*, ACM (2010).
7. Eilemann S., Makhinya M. & Pajarola R. Equalizer: a scalable parallel rendering framework. *IEEE TVCG 15*, 3 (2009).
8. Endert A., Andrews C., Lee Y. H. & North C. Visual encodings that support physical navigation on large displays. *GI '11*, CHCCS (2011).
9. Forlines C., Vogel D. & Balakrishnan R. Hybridpointing: fluid switching between absolute and relative pointing with a direct input device. *UIST '06*, ACM (2006).
10. Gjerlufsen T., Klokmose C., Eagan J., Pillias C. & Beaudouin-Lafon M. Shared Substance: developing flexible multi-surface applications. *CHI '11*, ACM (2011).
11. Greenberg S. Toolkits and interface creativity. *Multimedia Tools Appl. 32*, 2 (2007).
12. Greenberg S., Gutwin C. & Roseman M. Semantic telepointers for groupware. *OZCHI '96*, IEEE (1996).
13. Hachet M., Decle F., Knödel S. & Guitton P. Navidget for 3D interaction: camera positioning and further uses. *IJHCS 67*, 3 (2009).
14. Hutterer P. & Thomas B. H. Groupware support in the windowing system. *AUIC '07*, ACS (2007).
15. Isenberg P., Carpendale S., Bezerianos A., Henry N. & Fekete J.-D. Coconuttrix: collaborative retrofitting for information visualization. *IEEE CGA 29*, 5 (2009).
16. Jansen Y., Dragicevic P. & Fekete J.-D. Tangible remote controllers for wall-size displays. *CHI '12*, ACM (2012).
17. Jetter H.-C., Zöllner M., Gerken J. & Reiterer H. Design and implementation of post-wimp distributed user interfaces with ZOIL. *IJHCI 28*, 11 (2012).
18. Johanson B., Hutchins G., Winograd T. & Stone M. PointRight: Experience with flexible input redirection in interactive workspaces. *UIST '02*, ACM (2002).
19. Khan A., Fitzmaurice G., Almeida D., Burtnyk N. & Kurtenbach G. A remote control interface for large displays. *UIST '04*, ACM (2004).
20. McCallum D. C. & Irani P. ARC-Pad: absolute+relative cursor positioning for large displays with a mobile touchscreen. *UIST '09*, ACM (2009).
21. Myers B. A. Using handhelds and PCs together. *CACM 44*, 11 (2001).
22. Myers B. A., Bhatnagar R., Nichols J., Peck C. H., Kong D., Miller R. & Long A. C. Interacting at a distance: measuring the performance of laser pointers and other devices. *CHI '02*, ACM (2002).
23. Nacenta M. A., Pinelle D., Stuckel D. & Gutwin C. The effects of interaction technique on coordination in tabletop groupware. *GI '07*, ACM (2007).
24. Nancel M., Chapuis O., Pietriga E., Yang X.-D., Irani P. & Beaudouin-Lafon M. High-precision pointing on large wall displays using small handheld devices. *CHI '13*, ACM (2013).
25. Nancel M., Wagner J., Pietriga E., Chapuis O. & Mackay W. Mid-air pan-and-zoom on wall-sized displays. *CHI '11*, ACM (2011).
26. Olsen Jr. D. Evaluating user interface systems research. *UIST '07*, ACM (2007).
27. Olwal A., Frykholm O., Groth K. & Moll J. Design and evaluation of interaction technology for medical team meetings. *INTERACT '11*, Springer-Verlag (2011).
28. Pietriga E., Huot S., Nancel M. & Primet R. Rapid development of user interfaces on cluster-driven wall displays with jbricks. *EICS '11*, ACM (2011).
29. Rekimoto J. Pick-and-drop: a direct manipulation technique for multiple computer environments. *UIST '97*, ACM (1997).
30. Rohs M. Real-world interaction with camera phones. *UCS'04*, Springer-Verlag (2005).
31. Satyanarayan A., Weibel N. & Hollan J. Using overlays to support collaborative interaction with display walls. *IUI '12*, ACM (2012).
32. Scott S. D., Sheelagh M., Carpendale T. & Inkpen K. M. Territoriality in collaborative tabletop workspaces. *CSCW '04*, ACM (2004).
33. Stewart J., Bederson B. B. & Druin A. Single display groupware: a model for co-present collaboration. *CHI '99*, ACM (1999).
34. Stoakley R., Conway M. J. & Pausch R. Virtual reality on a WiM: interactive worlds in miniature. *CHI '95*, ACM/Addison-Wesley (1995).
35. Tse E. & Greenberg S. Rapidly prototyping single display groupware through the sdgoolkit. *AUIC '04*, ACS (2004).
36. Tse E., Histon J., Scott S. D. & Greenberg S. Avoiding interference: how people use spatial separation and partitioning in sdg workspaces. *CSCW '04*, ACM (2004).
37. Vogel D. & Balakrishnan R. Distant freehand pointing and clicking on very large, high resolution displays. *UIST '05*, ACM (2005).
38. Wigdor D., Jiang H., Forlines C., Borkin M. & Shen C. Wespace: the design development and deployment of a walk-up and share multi-surface visual collaboration system. *CHI '09*, ACM (2009).
39. Yee K.-P. Peephole displays: pen interaction on spatially aware handheld computers. *CHI '03*, ACM (2003).